割り算回路設計，あの手この手

デバイスの記事

最終回 5

テーブル近似と精度向上のための補正

関連データ

鈴木昌治

ここでは，テーブルを用いて除算回路を実現する方法を解説する．テーブルを使用しても，補正を加えることである程度の精度を維持することができる．なお，本稿で設計した回路のソース・コードは本誌付属CD-ROMに収録している．（編集部）

除算回路を過去4回にわたって説明してきましたが，今回で本連載も終わりです．

連載第1回（本誌2005年8月号，pp.109-117）と第2回（同年10月号，pp.136-144）で述べた回復法と非回復法は，けた単位で商を確定していく方法でした．また，第3回（同年11月号，pp.100-109）ではその高速化について述べました．第4回（同年12月号，pp132-138）は漸近法により，けたを超えて真の商に近づける方法を紹介しました．いずれも「繰り返し演算が必要である」という点で共通していました．

今回はテーブル近似により，「繰り返し演算を伴わずに商に直接たどり着く」方法を取り上げましょう．テーブル＝メモリという割り当てを考えたとき，メモリが潤沢で使いやすくなった近年のFPGAにおいて非常に有効な方法であるとともに，除算だけではなく初等超越関数（三角関数や対数関数）の実現にも応用できるので，ぜひとも活用していただきたいと思います．

「テーブルでは精度が今ひとつ」という批判を聞いたことがあります．おそらく商をすべてテーブル索引だけで処理しようとするからでしょうが，それではあまりに芸がありません．各種補正を加えるといったくふうを施すことで，精度を上げられます．

● 除算ではなく逆数関数で実現

あえて述べるまでもないことですが，除算は除数と被除数という二つの引き数が必要となる演算です．テーブルを用いて除算を実現する場合で，二つの引き数の組み合わせとなると，どうしてもテーブルが肥大化してしまい，現実的ではありません．このあたりの事情は，前回述べたNewton-Raphson法と似ています．そこで，今回もNewton-Raphson法の場合と同じく，逆数関数を実現することにします．

連載第2回のコラムで，「除算は浮動小数点と相性が良い演算である」と述べましたが，ここではその特性を利用しましょう．というのも，かりに入力である分母dを正数に限ったとしても，すべての範囲をテーブルでカバーするのは不可能だからです．有限なテーブル上で効率良く関数を表現するには，逆数の場合，浮動小数点表現を用いて次式の範囲に入力を限定することです．

$$1 \leqq d < 2 \quad \cdots\cdots (1)$$

これにより分解能を決定すれば，テーブルの深さ（アドレス幅）を決められます．また，式(1)によって商qは，式(2)のようになります．

$$0.5 < q \leqq 1 \quad \cdots\cdots (2)$$

要求されるビット精度が決まれば，テーブルのデータ幅を決定できます．

以上で，テーブルを用いた逆数演算の土台ができ上がりました．

1 基本は直線補間

d，qともに8ビット程度の精度が要求される場合，2Kビットのメモリが必要となります．今どきは，すべてテーブル化してしまったほうが現実的でしょう．しかし，16ビッ

トの精度が入出力に求められる場合には，メモリは1Mビットとなり，ASICやFPGAに内蔵するテーブルとしては大きなものになります(除算の前半部でしかない逆数演算に許されるサイズではない)．

ここで登場するのが直線補間による補正方法です．定義域をいくつかの区間に分割し，その境界における関数値をテーブル上に持ち，一つの区間の値を直線補間による演算で求める方法です．

● 直線補間を用いると回路はシンプル，精度はいまいち

図1に，直線補間の原理を示します．ここでは見やすくするために二つの区間に分割し，$1 \leqq d < 1.5$ の区間を直線補間しているようすを表しています．求める値を $d = 1 + k$ のときの値とすれば，$f'(1+k)$ は式(3)で与えられます．

$$f'(1+k) = (1-k) \times f(1) + k \times f(1.5) \quad \cdots\cdots\cdots(3)$$

まずは，入出力16ビットとして話を進めましょう．**図1**のように式(1)の区間を2分割というわけにはいかないので，ここでは32分割することにします．テーブルのアドレスは5ビットになります．式(1)より d のMSB(most significant bit)はつねに'1'なので，これを除外した小数点以下の部分を16ビットに割り当てます．そのため，[15：11]がテーブルを引くための5ビットになります．ここで，1区間の幅を h とすると次式のようになります．

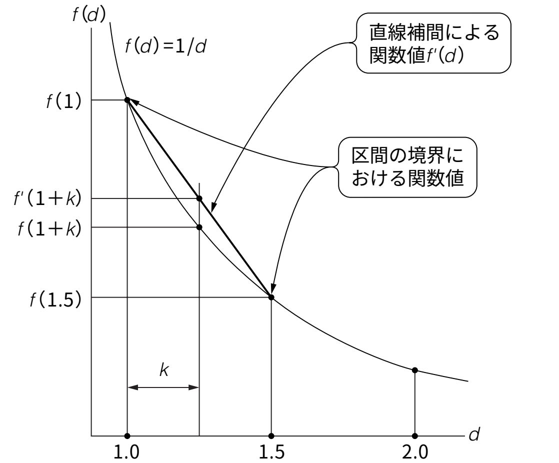


図1　直線補間による逆数演算の原理
$1 \leqq d < 2$ の正規化区間をさらに分割し，各区間に直線補間を適用する．

$$h = 1/32 \quad \cdots\cdots\cdots(4)$$

出力の16ビット精度を目標としているので，テーブル上の境界点データは2ビットほど余裕をみて18ビットとします．したがって，32区間×18ビット×2(区間の始点と終点の2データが必要)で1Kビット強のテーブルとなります．

テーブルから出力された区間の始点と終点データを，d[10：0](テーブルを引くのに使用しなかった部分)で直線補間し，丸めを行えば演算完了です．

ブロック図を**図2**に，演算のけたの割り当てを**図3**に示

コラム1　関数実現のための直線補間

本稿のような初等超越関数を実現する際に直線補間を用いる場合，本文の**図2**に示した構成をとる必要はありません(本文中では，あくまで原理的表現としてある)．その理由は，直線補間が，

- 補間する2点がテーブルで与えられる
- 定義域が限定された場合，変化率が穏やかである

という条件を持つからです．

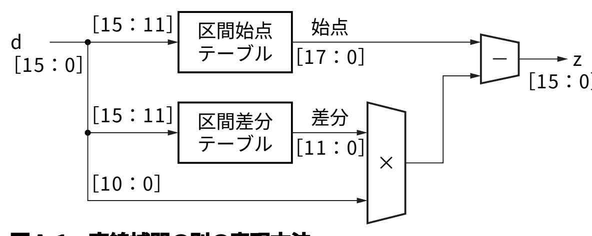
前者の理由から，無作為な同じディメンジョンを持つ2変数間を補間するのと異なり，直線補間ではその区間の始点と次の区間との差分を与えられれば補間可能ということになります(始点や終点は位置，差分は距離というディメンジョンの違いがある)．テーブルの中身は何でも可能ですし，関数そのものは既知のものなので，かならずしも始点と終点を与える必要はありません．これにより，始点を a，差分関数を $g(x)$ とすると，本文中の式(3)は次のように表現できます．

$$f'(a+k) = f(a) - k \times g(a) \cdots\cdots\cdots(\text{A-1})$$

明らかに演算が簡略化されています(逆数の場合は一様に減少する関数なので差分は減算されるが，増加する関数であれば加算となる．差分を符号付きとすれば加算で統一してもかまわない)．

一方，後者の理由からは(前者とも関連があるが)，差分テーブルは終点テーブルに比べてデータ幅を小さくできます．本文で示した例では，終点テーブルは18ビット幅でしたが，差分テーブルであれば12ビット幅ですむため，テーブル・サイズを小さくできます．当然，乗算器も小さい規模のものですみます．

これによって，**図2**は**図A-1**のように表現できます．



図A-1　直線補間の別の表現方法

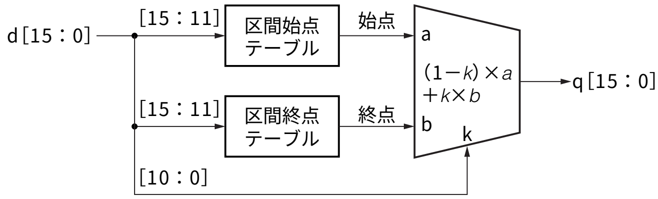


図2　直線補間による逆数演算ブロック図

*d*の上位けたでテーブルを引き，得られた区間の始点と終点を *d*の下位のけたで補間する．

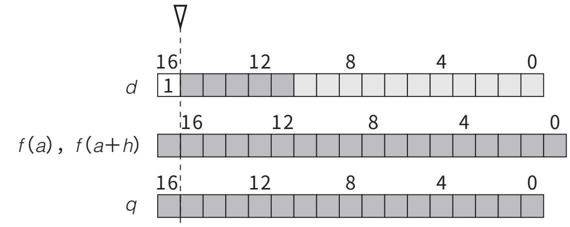


図3　直線補間による逆数演算のけた割り当て

f(*a*)と*f*(*a*＋*h*)は，それぞれテーブル出力の始点と終点のデータを表す．

リスト1　図2の回路のRTL記述

```
module  div_table_a (
  q   , // quotient  [16]:int [15:00]:fraction
  d ); // divisor  [15:00]:fraction([16]:static int 1)

// io
  input  [15:00] d ;
  output [16:00] q ;

// function （演算処理部）
  // table
  function  [35:00] ftable ;
                        // テーブル [35:18]:始点 [17:00]:終点
     input  [15:11] d      ;
     case   ( d )
            5'h00   :   ftable = { 18'h20000,18'h1F07C } ;
            5'h01   :   ftable = { 18'h1F07C,18'h1E1E2 } ;
            5'h02   :   ftable = { 18'h1E1E2,18'h1D41D } ;

            ～中略～

            5'h1d   :   ftable = { 18'h10C97,18'h10842 } ;
            5'h1e   :   ftable = { 18'h10842,18'h10410 } ;
            5'h1f   :   ftable = { 18'h10410,18'h10000 } ;

    endcase
  endfunction

  wire  [17:00]  p_stt  ;
  wire  [17:00]  p_end  ;
  assign        {p_stt,p_end} = ftable (d[15:11] ) ;

  // bi-linear interpolation （直線補間）
  wire  [29:00]  intrp = ( 12'h800 - d[10:00] ) * p_stt
                         + d[10:00] * p_end + 12'h800 ;

  assign         q = intrp[28:12] ;

endmodule
```

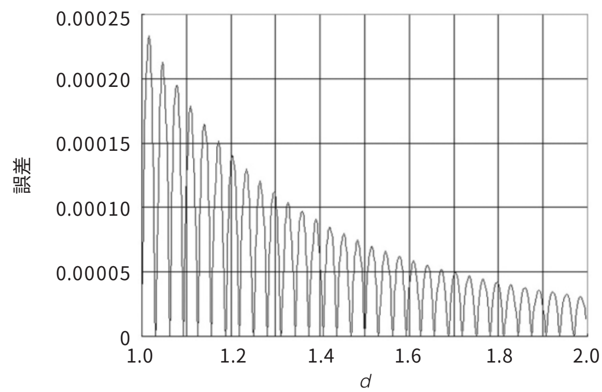


図4　直線補間による逆数演算の誤差

*d*が1に近いほど誤差が大きいのは，より曲率が大きい（直線になじみにくい）からである．

します．また，この回路のRTL記述を**リスト1**に示します．出力のMSBは$d=1$のとき以外はつねに'0'なので，実質の16ビットという意味で出力は17ビットとしてあります（原理に忠実に表現しているので，実際の構成については，p.129のコラム「関数実現のための直線補間」を参照）．

図2を見てわかるのは，非常にシンプルな構成であるということです．これが，直線補間を用いる最大のメリットでしょう．反面，精度はそれほど高くないというデメリットもあります．誤差を式(5)のように定義します．

$$\varepsilon = f'(d) - f(d) \quad \cdots\cdots(5)$$

定義域における誤差のようすを**図4**に示します．各区間の中央に誤差のピークがあり，それが分割した区間の数だけ並んでいます．dが1に近いほど誤差が大きいのは，より曲率が大きい（直線になじみにくい）からです．

結果的には，16ビット精度を目指しながらも12ビット程度の精度しか得られていないことがわかります．ここで注意してほしいのは，**図4**はRTLのシミュレーション結果を表しているわけではなく，米国Microsoft社のExcelでシミュレーションしたものだということです．Excelでは10進で15けたの精度があるということなので，テーブルや補間演算などはかなりの精度で演算されています．つまり，直線補間を用いる限り，テーブルや演算の精度を上げたところで，これ以上は望めないわけです．では，どうすれば精度を上げられるのでしょうか．

● 区間を細分化して精度を上げる

図1から，「区間をもっと細かくすれば精度を上げられる」ということが直感的に理解できると思います．しかし，やみくもに試行錯誤で精度を上げていくのは，スマートな方法とはいえません．まず，区間の分割数と精度の関係を

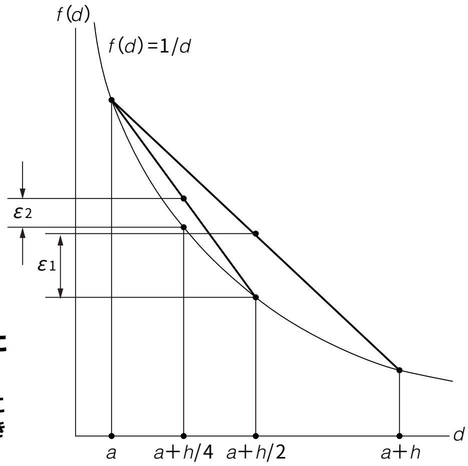


図5 区間分割の違いによる誤差の推定
区間を細分化することで精度を向上できる．

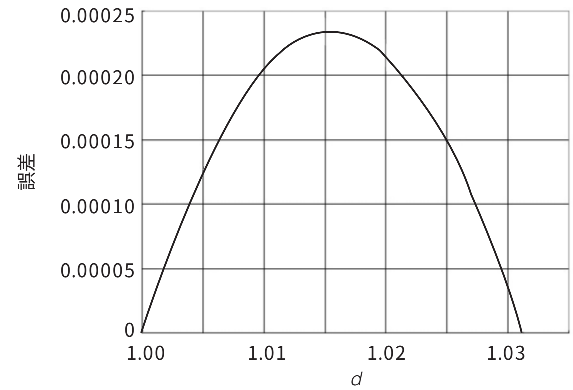


図6 直線補間による最初の区間の誤差
誤差のピークが中央にあって，ほぼ左右均等の誤差を示していることから，偶数次成分が大きいことがわかる．

考察してみましょう．

図5に区間分割の違いによる誤差の違いを示します．区間の始点をa，区間の幅をhとします．誤差の最大値が区間の中央にあると仮定するとき，最大誤差ε_1は，式(6)で与えられます．

$$\varepsilon_1 = \frac{1}{a+h/2} - \left(\frac{1}{a} + \frac{1}{a+h}\right)/2 \quad \cdots\cdots(6)$$

区間の幅が$h/2$となった場合の最大誤差は，式(7)で与えられます．

$$\varepsilon_2 = \frac{1}{a+h/4} - \left(\frac{1}{a} + \frac{1}{a+h/2}\right)/2 \quad \cdots\cdots(7)$$

以上より，区間が1/2となった場合(区間の分割数を2倍とした場合)の誤差の比は，次式で表されます．

$$\frac{\varepsilon_2}{\varepsilon_1} = \frac{a+h}{4a+h} \quad \cdots\cdots(8)$$

誤差が最大となる区間は$d=1$を始点とする最初の区間であることはわかっています．そこで$a=1$として，hが分割を進めるほど微小区間となることを考慮して，$h \to 0$の極限をとります．

$$\lim_{h\to 0}\frac{\varepsilon_2}{\varepsilon_1} = \lim_{h\to 0}\frac{h+1}{h+4} = \frac{1}{4} \quad \cdots\cdots(9)$$

式(9)から，区間の分割数を2倍にすると精度は4倍になることがわかります．実際には$h=0$ではないので，4倍よりは若干小さな値になりますが，およその目安にはなるでしょう．

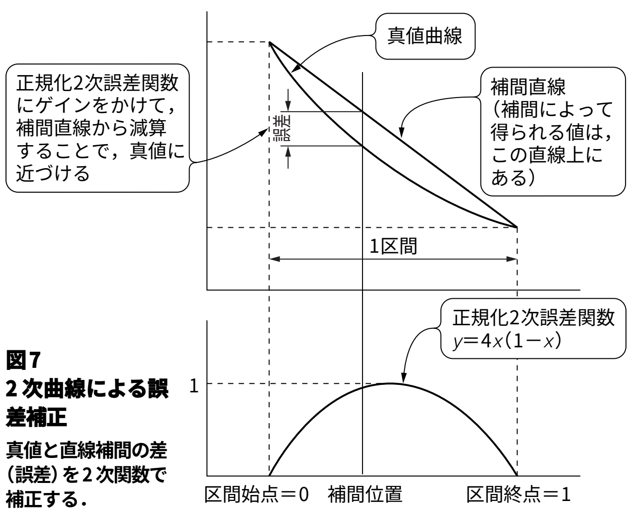


図7 2次曲線による誤差補正
真値と直線補間の差(誤差)を2次関数で補正する．

先ほどの32分割では12ビットの精度だったので，16ビットに追い込むには4倍の128分割とすればよいことがわかります．この程度であれば許されるテーブル・サイズですが，さらに20ビット，24ビットと高い精度が必要とされた場合には規模的に厳しくなってきます．また，精度を上げるので，データ幅も大きくとらなくてはならないことが，これに拍車をかけることになります．分割数を上げるだけではおのずと限界が見えてきます．

2 高次補正を採用して精度を上げる

直線補間を級数展開などにおける1次までの近似ととらえるならば，2次的な補正を加えることで精度を上げられるということが考えられます．初等超越関数を級数展開した場合，高次になるほど係数が小さくなるのが一般的なの

で，「1次の次は2次」というのはきわめて自然な考えかたでしょう注1.

● 2次補正で精度を20ビットに向上

図6に直線補間（32分割）の場合の最初の区間における誤差を拡大して示します．誤差のピークが中央にあって，ほぼ左右均等の誤差を示していることから，偶数次成分が大きいことがわかります．この形であれば，2次の補正をかければ精度を向上させられることが予想できます．そこで，

$$u = 4t(1-t) \quad \cdots\cdots(10)$$

という2次関数を定義し，分割された1区間を$0 \leqq t < 1$に割り当てます（**図7**）．つまり，直線補間に用いたd［10：0］をこの範囲に見立てるわけです．そして，**図7**を正規化された2次誤差補正関数と考え，区間ごとにゲインを変えて直線補間値から減算してやればよいのです．2次誤差補正を施した後の誤差を**図8**に示します（Excelによるシミュレーション結果）．この時点での精度は20ビット弱まで上がるので，初期の要求仕様は満たされたことになります．

要求精度を20ビットとしたうえで，ブロック図を**図9**に，演算のけたの割り当てを**図10**に，またRTL記述を**リスト2**に示します．

図11に，最初の区間における誤差を拡大して示します．この図からわかることは，1区間内の誤差が奇数次成分によって占められていることです．直線補間によって1次成分を，2次補正によって2次成分をそれぞれ除去したので，3次成分が支配的になるのは先の級数展開の例を出すまでもなく直感的に理解できるでしょう．さらなる精度を期待するならば，次は3次成分を除去すればよいのですが，演

注1：逆数の場合，$d=0$でマクローリン展開できない．しかし，式(1)のように定義域を決めているので，関数を$1/(1+x)$として定義域を$0 \leqq x < 1$としてやれば，級数展開も可能になる．もっとも，マクローリン展開などは局所的にしか精度が得られないので，汎用性はない．

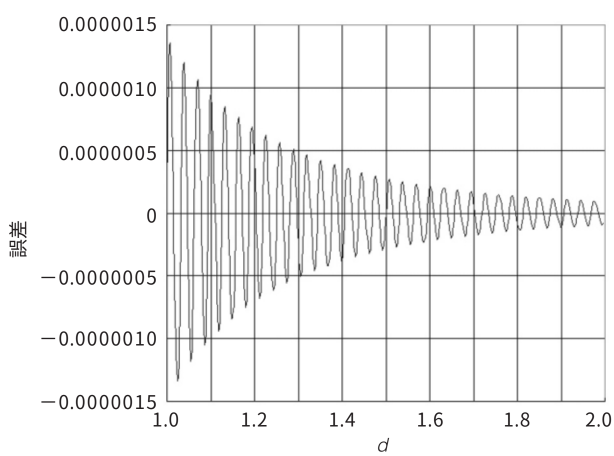


図8　2次補正による逆数演算の誤差
精度が20ビット弱まで上がり，初期の要求仕様は満たされたことになる．

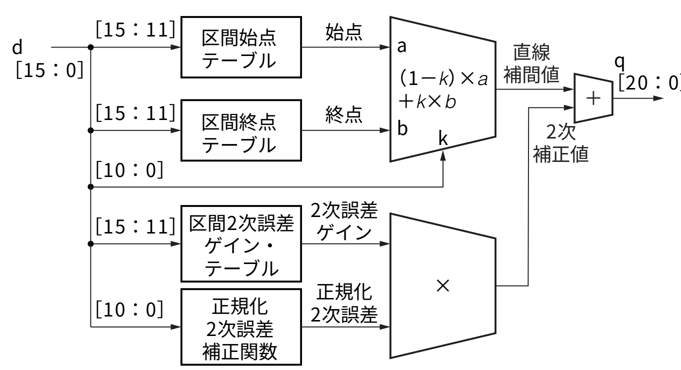


図9　2次補正追加による逆数演算ブロック図
直線補間による回路に2次誤差補正機能を追加し，演算精度を向上させる．

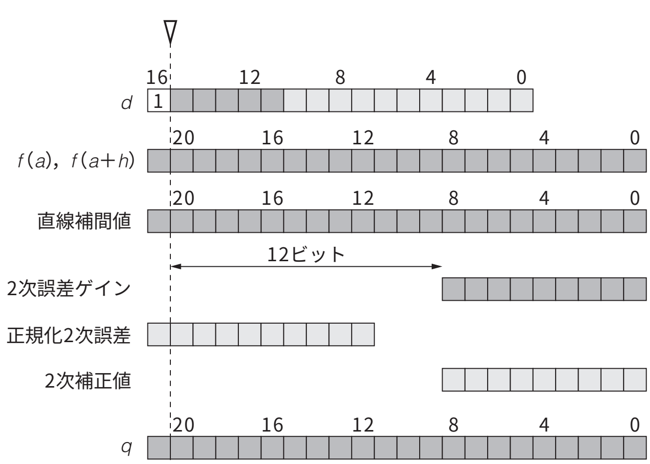


図10　2次補正追加による逆数演算のけた割り当て
精度向上のため，けたを追加した．

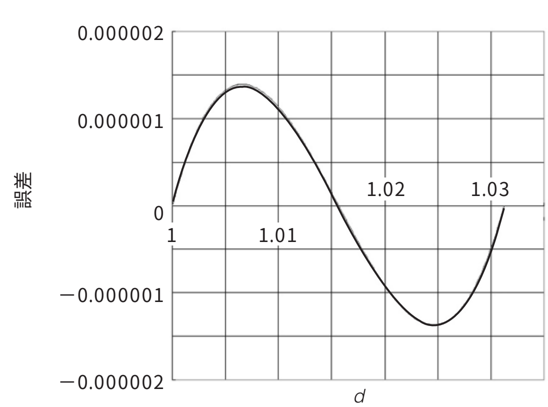


図11　2次補正による最初の区間の誤差
1区間内の誤差が奇数次成分によって占められていることがわかる．

算の次数が高くなると当然，演算回路がかさばってきます．なにか，逃げ道はないのでしょうか．

● さらに2次以上の補正をテーブル化

式(10)は2次関数なので，これで補正をかけると2次成分にしか効かないのは明らかです．せっかくテーブルを利用しているのですから，2次以上の成分をまとめてテーブル化して，式(10)の代わりに使用すれば，高次成分も除去できるはずです．この方法を検討してみましょう．

ここで，「だったら1次成分もテーブルに持ち込んで，まとめて補正してしまえばよいのでは」と考える方もいるかもしれません．考えかたとしてはまちがっていないのですが，1次成分は2次以上の成分と比べ非常にレベルが高いので，テーブル自体がとても大きくなってしまうのです[注2]．

補正テーブルを区間ごとに持てば，非常に高い精度を得られますが，1区間のアドレスだけでも11ビットあるので，これをすべての区間に持つことは避けることにしましょう．1区間の正規化された代表値をテーブル化し，先の2次補正の場合と同じように区間ごとのゲインを変えて，直線補間値から減算することにします．代表値には，直線補間による最初の区間の誤差を正規化して用います．

この補正方式による誤差を**図12**（Excelによるシミュレーション結果）に，ブロック図を**図13**に，演算のけたの割り当てを**図14**に示します．なお，**リスト1**と**リスト2**に基本的な構造が示されているので，この回路のRTL記述は割愛します．

図12から，23ビット弱の精度を得ることが確認できま

注2：2次までは演算で補正をかけて3次以上をテーブルで処理したほうが，テーブルを小さくできる．本稿では，演算回路を肥大化させないという論旨でここにたどり着いているので，2次成分は含めることにする．

リスト2　図9の回路のRTL記述

```
module div_table_b (
  q  , // quotient [20]:int [19:00]:fraction
  d ); // divisor [15:00]:fraction([16]:static int 1)

// io
  input  [15:00] d ;
  output [20:00] q ;

// function (演算処理部)
  // table
  function  [52:00]  ftable  ;
                       // [52:31]:始点 [30:09]:終点 [08:00]:ゲイン
     input  [15:11]  d       ;
     case   ( d )
            5'h00   :   ftable = { 22'h200000,22'h1F07C2,9'h1E9 } ;
            5'h01   :   ftable = { 22'h1F07C2,22'h1E1E1E,9'h1BE } ;
            5'h02   :   ftable = { 22'h1E1E1E,22'h1D41D4,9'h199 } ;
            ～中略～
            5'h1d   :   ftable = { 22'h10C971,22'h108421,9'h048 } ;
            5'h1e   :   ftable = { 22'h108421,22'h104104,9'h045 } ;
            5'h1f   :   ftable = { 22'h104104,22'h100000,9'h042 } ;
    endcase
  endfunction

  wire  [21:00]  p_stt ;
  wire  [21:00]  p_end ;
  wire  [08:00]  gain ;
  assign        {p_stt,p_end,gain}
                             = ftable (d[15:11]);

  // interpolation
  wire  [33:00]  intrpa = ( 12'h800 - d[10:00] )
       * p_stt + d[10:00] * p_end ;     // 直線補間
  wire  [20:00]  err2st = ( 12'h800 - d[10:00] )
           * d[10:00] + 11'h400 ;   // 正規化2次関数
  wire  [17:00]  intrpb = err2st[20:11] * gain ;
                                        // 2次誤差生成
  wire  [21:00]  intrp
       = intrpa[33:11] - intrpb[17:09] + 1'b1 ;
                                   // 直線補間－2次誤差
  assign         q      = intrp[21:01] ;

endmodule
```

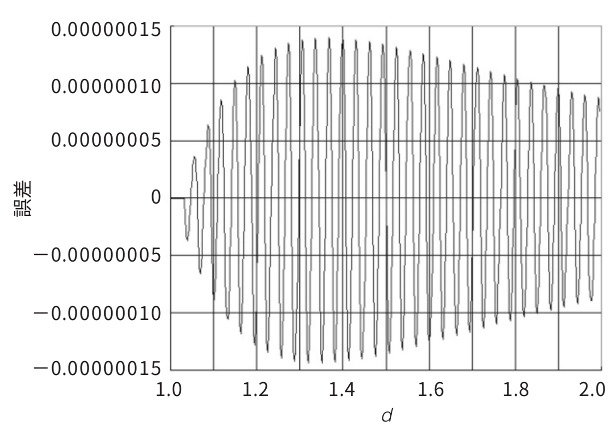


図12　高次補補正による逆数演算の誤差

23ビット弱の精度を得ることが確認できるが，期待された飛躍的な精度の向上には至っていない．

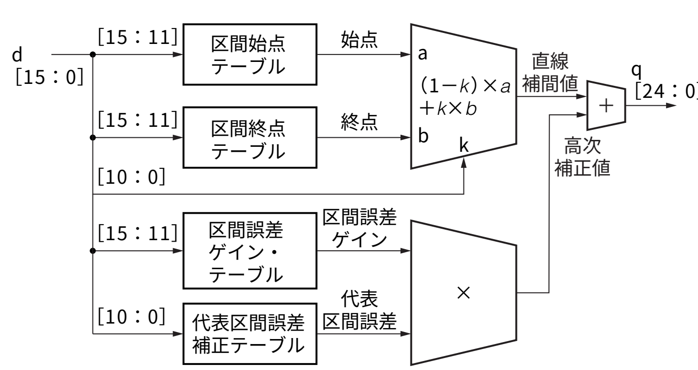


図13　高次補正追加による逆数演算ブロック図

2次関数による補正回路の代わりに，テーブルによる補正回路を追加し，精度を向上させている．

すが，期待された飛躍的な精度の向上には至りませんでした．「すべての次数を含んでいるテーブルを用いたのだから，演算誤差以外は除去されるのではないか」と期待された方もいるでしょう．この原因は，直線補間による最初の区間の誤差を，代表値としてすべての区間に用いたことにあります．実際には各区間における高次成分の比は区間ごとに違っているのです．したがって，ゲインを区間ごとに調整しても，そのゲインは支配的な2次成分に合わせることになるため，3次以上の成分に対しては不完全な補正となってしまうのです．最初の区間は見事に誤差が除去できていること，そのほかの区間では3次成分が支配的であることから，そのようすがよくわかります．

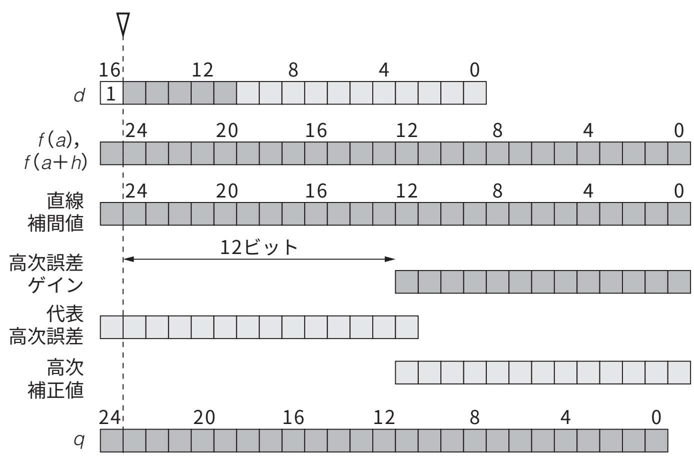


図14　高次補正追加による逆数演算のけた割り当て
精度向上のため，けたを追加した．

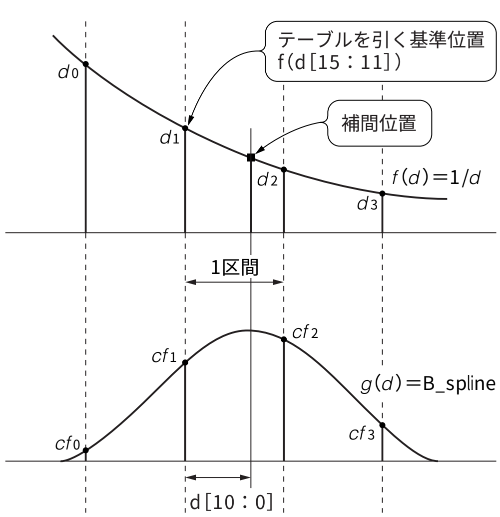


図15　B_Splineによる補間原理
ディジタル・フィルタと同じように畳み込みを行う．1区間の分解能分の係数が必要．

● 補間関数の高次化

ここまでは，最初に直線補間ありきで，そこで取りきれなかった高次成分をどう除去するかという視点で考えてきました．逆数関数は曲線なので，直線補間では取りきれない成分がかならず付きまとっていたからです．次は，少し視点を変えてみましょう．

ディジタル信号処理における補間処理では，直線補間はきわめてベーシックな手段です．より良い結果を求める場合は，次数の高いディジタル・フィルタが使用されます．「区間ごとの離散的な代表値をそれらしく補間する」という視点に立てば，関数の波形を再現することにも応用できそうです．ここでは，補間関数として3次のB_Splineを採用しましょう（右掲のコラム「B_Spline関数」を参照）．

3次B_Splineの補間原理を**図15**に示します．**図15**において，補間位置での関数値は式(11)で表されます．

$$f(d)=\sum_{k=0}^{3} d_k \times cf_k \quad \cdots\cdots(11)$$

この式(11)の演算を回路化すればよいわけですから，ここまで使用した区間の始点/終点テーブルの代わりに，d_0～d_3の値を区間ごとにテーブルとして持つことになります．もっともd_1とd_2は，区間の始点と終点とまったく同

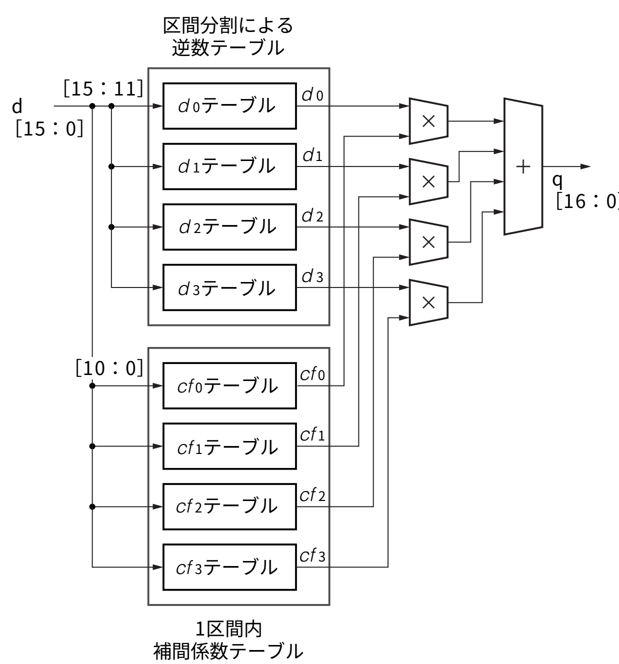


図16　B_Spline補間による逆数演算ブロック図
*d*の上位で区間ごとの逆数テーブル4点を引き，下位で係数テーブルを引いて畳み込みを行う．

じものですし，d_0 と d_3 は1区間隣の始点と終点になります．どういうテーブルが必要かはあえて説明するまでもないでしょう．さらに，新しい要素としてB_Spline関数の発生回路が必要になります．数値演算で発生させるのもよいのですが，話の流れからはテーブルでの表現となるでしょうか．このあたりは，状況に応じて判断しましょう．

図16に，ブロック図を示します．B_Spline関数はテーブルで表現してあります．乗算器が四つ並んでいるのが回路規模の面からは気になるところですが，構造が単純なので4サイクルかけて1系統で済ませることも可能です．誤差の評価を**図17**に示します．補正などを加えずに，この時点で16ビット弱の精度が確保されています．また，1区間内の誤差に着目すれば，1次成分が支配的なのは明白ですから，補助的な補正をかけることで精度の向上が十分期待できます．

＊　＊　＊

今回紹介したテーブルによる手法は，あくまで1例と考えてください．要求される精度，リソース，遅延（レイテンシ）など状況に応じて，実現する形はどのようにでもなるということです（テーブルと演算の分担などは，チューニングによって回路規模に大きな影響を与える）．また，ほかの手法を組み合わせることでおもしろい結果が得られる可能性もあります．柔軟な発想で新しい実現方法を模索してみるのも楽しいでしょう．

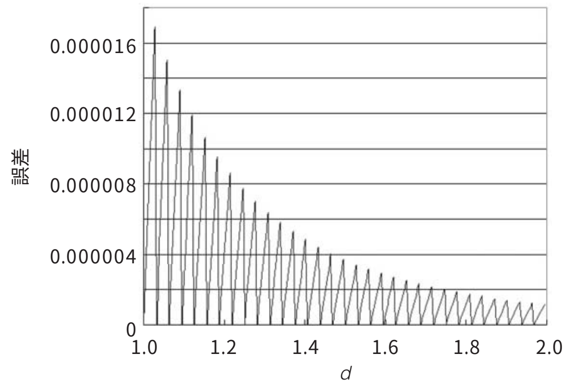


図17　B_Spline補間による逆数演算誤差
補正などを加えずに，この時点で16ビット弱の精度が確保されている．

また，これらのテーブル手法が逆数関数に限られたものではないことはすでにお気づきでしょう．テーブルをROMでなくRAMで実現し，外部から内容を書き換えることでほかの関数も実現できます（関数の特性がそれぞれ違うので，若干冗長性が必要になると思うが…）．

すずき・しょうじ
スタンダード・ロジック（株）

<筆者プロフィール>
鈴木昌治．枯れた回路と思われる部分でも，何かおもしろくならないかと，日々，回路設計を楽しんでいます．もちろん，「顧客や上司ににらまれない範囲で」ですが….

コラム2　B_Spline関数

Spline（スプライン）関数は，離散的な点を適当な滑らかさで連結させる関数です．平面上の離散点を自在定規で結ぶ操作を数学的に解決する方法と考えることができます．

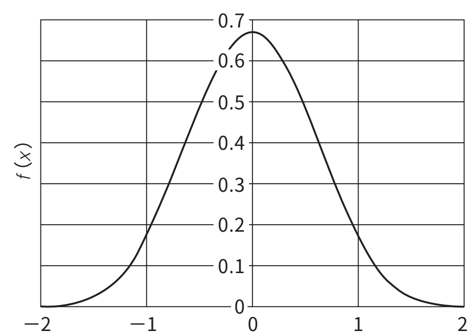
スプライン関数の用途は多岐に渡りますが，関数の近似も有効なアプリケーションの一つです．本稿では，テーブル上の各区間の境界点をそれらしく結ぶことで区間内を補間し，関数を近似するために使用しています．

B_Splineは，スプライン関数の中でもっとも基本的（basic）な関数であり，3次の場合は以下のように定義されます．

$$f(x)=\begin{cases}\frac{1}{2}|x|^3-|x|^2+\frac{2}{3} & (0\le|x|<1)\\ -\frac{1}{6}|x|^3+|x|^2-2|x|+\frac{4}{3} & (1\le|x|<2)\\ 0 & (2\le|x|)\end{cases} \quad \cdots\cdots\cdots(\text{B-1})$$

図B-1は，その関数波形を示しています．

式（B-1）が与えられていれば，補間係数テーブルを作成できるので，スプライン関数について詳細を知っていなくても関数の近似を行うことができます．もちろん，テーブルを用いずに演算で補間係数を求めることもできますね．



図B-1
B_Spline関数